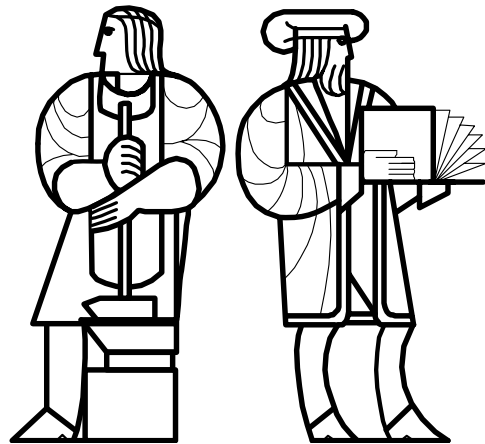


Specifications



6.170

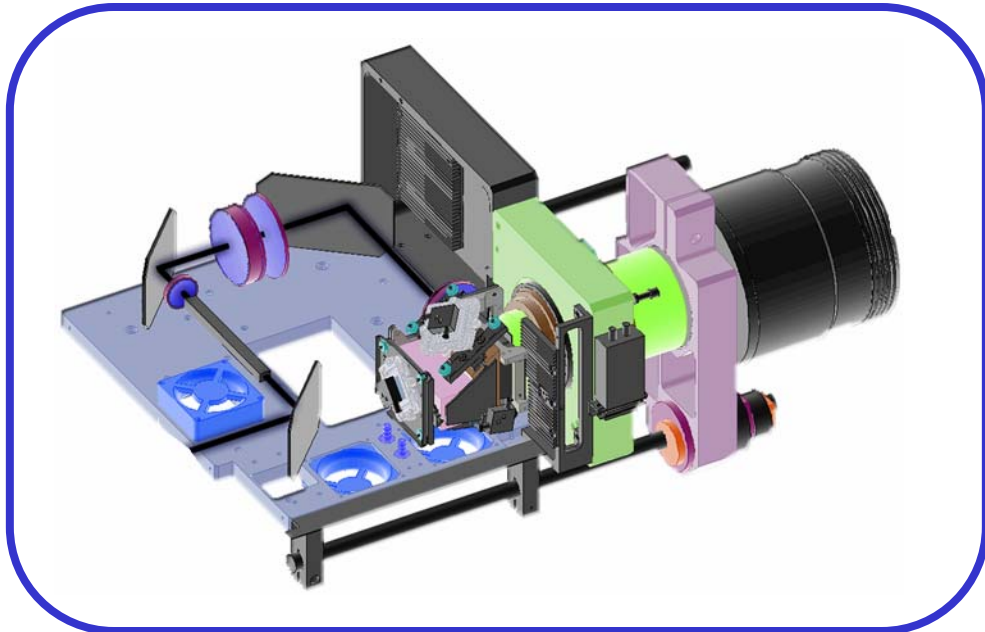
Michael Ernst

Saman Amarasinghe

Review of Classes

Why classes?

Encapsulation, data hiding, access control



Review of Classes

Why classes?

Encapsulation, data hiding, access control

```
class Account {  
    String name;  
    int balance = 0;  
    Account(String name) {  
        this.name = name;  
    }  
    public boolean check(Transaction t) {  
        return (balance + t.amount) >= 0;  
    }  
}  
  
Account acc = new Account("My Checking");
```

Members

Constructors

Methods

Instance

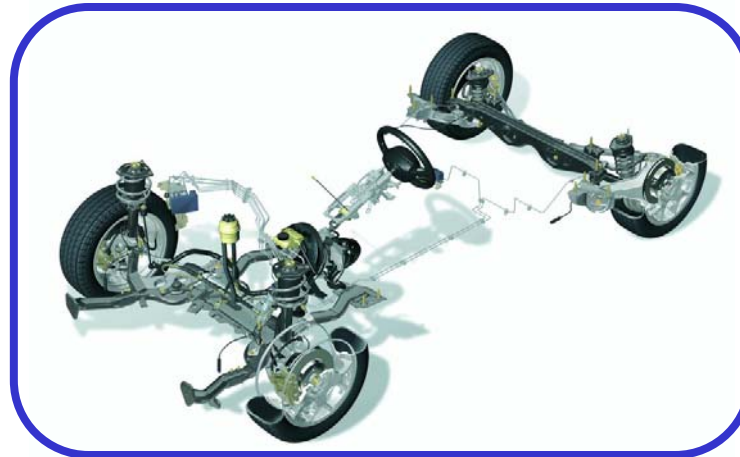
Access
Control

Review of Inheritance

Why classes?

Encapsulation, data hiding, access control

Inheritance (sharing)



Review of Inheritance

“Is-a” relationship

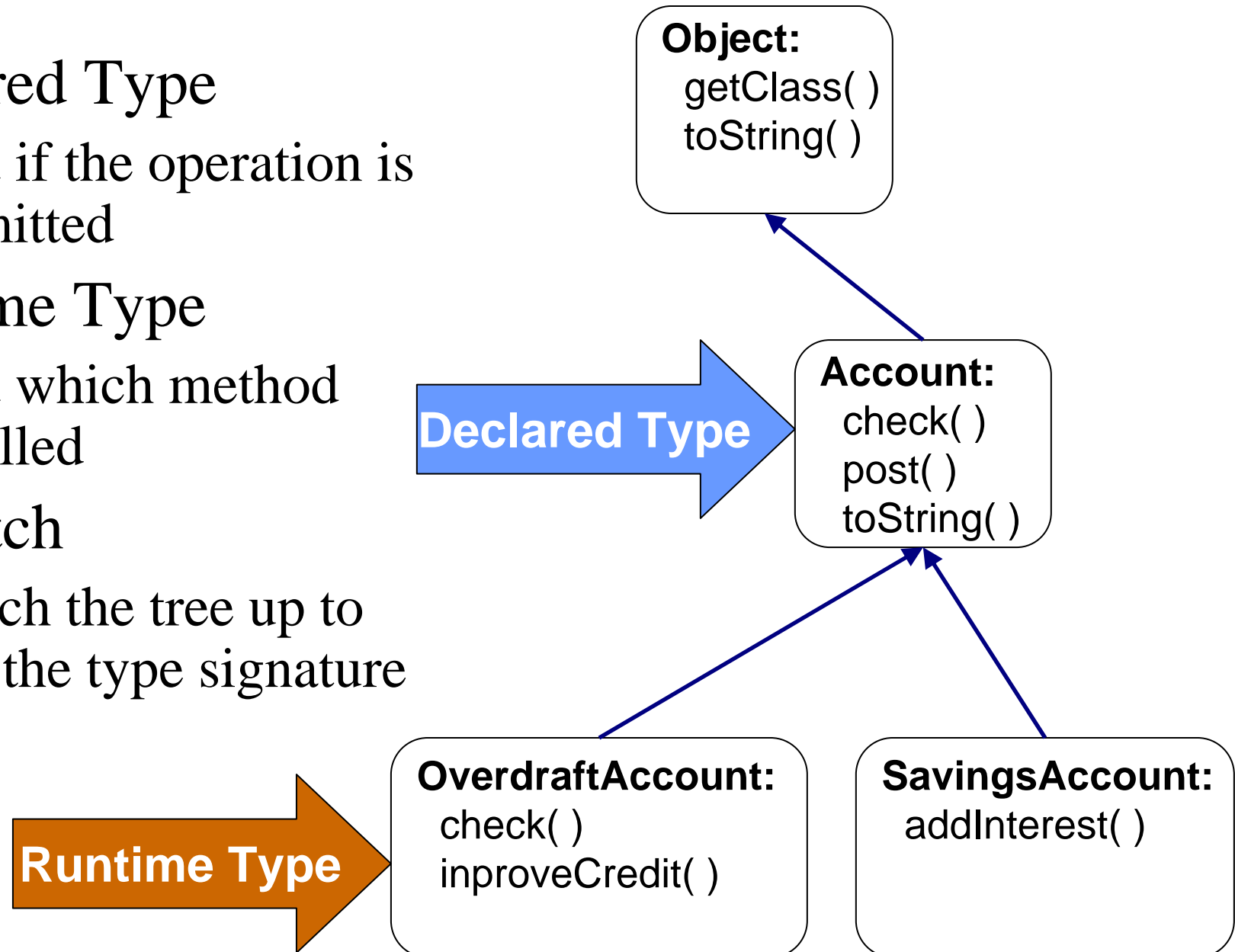
Subtype

Supertype

```
class OverdraftAccount extends Account {  
    int creditLimit;  
    boolean check(Transaction t) {  
        return (balance + t.amount) >= -creditLimit;  
    }  
}
```

Review of Declared and Runtime Types

- Declared Type
 - Find if the operation is permitted
- Runtime Type
 - Find which method is called
- Dispatch
 - Search the tree up to find the type signature



Review of Abstract Classes

An abstract class is partially implemented

Some declared methods are not implemented

The class cannot be instantiated

Subclasses inherit implementations, fields

A concrete subclass must implement all methods

```
abstract class Account {  
    String name;  
    int balance = 0;  
    Account(String name) {  
        this.name = name;  
    }  
abstract boolean check(Transaction t);  
}  
...  
class CheckingAccount extends Account ...  
...  
class SavingsAccounts extends Account ...
```



Review of Interfaces

An interface is not at all implemented

Methods may be declared

No fields, no constructor

```
interface List<E> {  
    public abstract int size();  
    public abstract boolean isEmpty();  
    public abstract void add(E);  
    public abstract int indexOf(E);  
    public abstract get(int);  
    public abstract remove(E);  
}
```

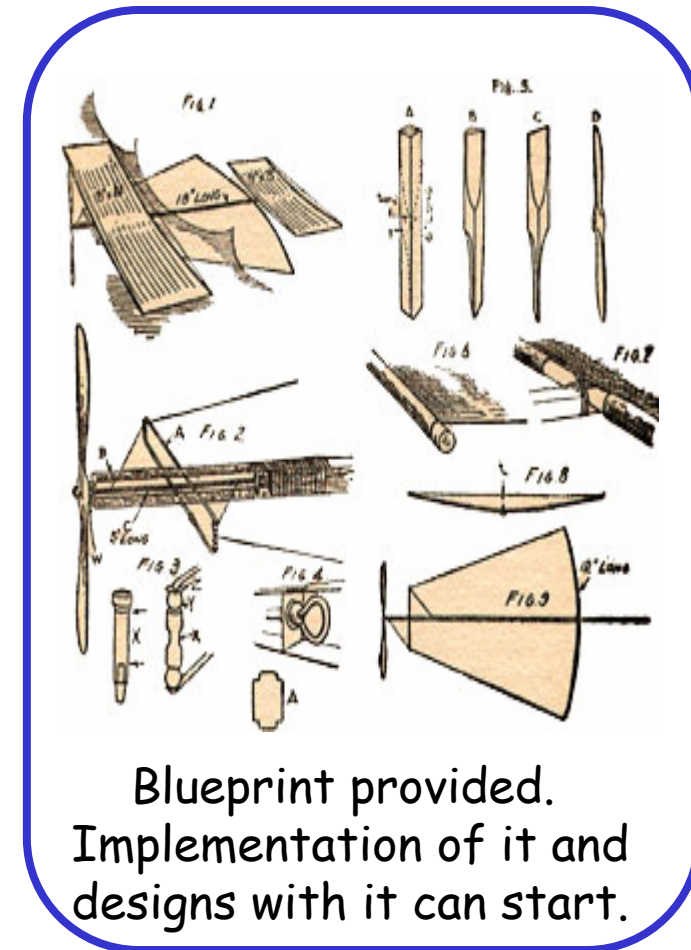
...

```
class LinkedList<E> implements List<E> ...
```

...

```
class ArrayList<E> implements List<E> ...
```

...



the challenge of scaling software

Small programs are simple and malleable

easy to write

easy to change

Big programs are (often) complex and inflexible

hard to write

hard to change

Why does this happen?

Because interactions become unmanageable

How do we keep things simple and malleable?

a useful discipline

We often view a program in two distinct ways:

The implementor's view (how to build it)

The client's view (how to use it)

It helps to apply these views to program *parts*:

While implementing one part, consider yourself a client of any other parts it depends on

Try *not* to look at those other parts through an implementor's eyes

This helps dampen interactions between parts

Formalized through the idea of a specification

what is a specification?

“Specification is the specific set of (high level) requirements agreed to by the sponsor/user and the manufacturer/producer of a system.” - Wikipedia

A specification is a contract

between client and implementor, describing their expectations of each other

Facilitates simplicity

Isolate client from implementation details

Isolate implementor from what use is made of the part

Discourages implicit, unwritten expectations

Facilitates change

Reduces the “Medusa” effect: the specification, rather than the code, gets “turned to stone” by client dependencies

Isn't the interface sufficient?

The main reason for the interface is to define the boundary between the implementers and users:

```
public interface List<E> {  
    public int get(int);  
    public void set(int, E);  
    public void add(E);  
    public void add(int, E);  
    ...  
    <T> public static boolean sub(List<T>, List<T>);  
}
```

Interface provides the syntax

But nothing on the effects and impact

why not just read code?

```
<T> boolean sub(List<T> src, List<T> part) {
    int part_index = 0;
    for (T elt : src) {
        if (elt.equals(part.get(part_index))) {
            part_index++;
            if (part_index == part.size()) {
                return true;
            }
        } else {
            part_index = 0;
        }
    }
    return false;
}
```

code is complicated

Code gives more detail than needed by client

For large program, understanding or even reading every line of code is an excessive burden

Suppose you had to read source code of Java libraries in order to use them

Same applies to developers of different parts of the libraries

Client cares only about what code does, not how it does it

code is ambiguous

Code seems unambiguous and concrete

But which details of code's behavior are essential, and which are incidental? This is a crucial ambiguity

Code invariably gets rewritten, so client needs to know what they can rely on

what properties will be maintained over time?

what properties might be changed by future optimization, improved algorithms, or just bug fixes?

Implementor also needs to know what features the client depends on, and which can be changed

what about code and comments?

Much of the code on the planet is decorated with short descriptions designed to convey the general idea of what that the code does:

```
// Check whether "part" appears as a  
// sub-sequence in "src".  
<T> boolean sub(List<T> src, List<T> part) {  
    ...  
}
```

Now, the client can often get along without reading the code, and if there's ambiguity they can just run a test to see what happens, right?

e.g. what if src and part are both empty lists

beyond the “general idea”

A description of a part becomes a specification if:

The client can and agrees to rely only on information in the description in their use of the part.

The implementor of the part promises to support everything in the description, but otherwise is perfectly at liberty

But this is not common behavior!

Clients often work out what a method/class does in ambiguous cases by simply running it, then depending on the results

This leads to programs with unclear dependencies, reducing simplicity and flexibility

Let's look at this code again...

```
<T> boolean sub(List<T> src, List<T> part) {
    int part_index = 0;
    for (T elt : src) {
        if (elt.equals(part.get(part_index))) {
            part_index++;
            if (part_index == part.size()) {
                return true;
            }
        } else {
            part_index = 0;
        }
    }
    return false;
}
```

a more careful description of sub()

```
// Check whether "part" appears as a  
// sub-sequence in "src".
```

needs to be given some caveats:

```
// * If src and part cannot be null  
// * If src is empty list, always returns false.  
// * Results may be unexpected if partial matches  
//   can happen right before a real match; e.g.,  
//   list (1,2,1,3) won't be found in (1,2,1,2,1,3).
```

or replaced with a more detailed description:

```
// This method scans the "src" list from beginning  
// to end, building up a match for "part", and  
// resetting that match every time that...
```

or reorganize to be simple

Complicated description suggests poor design

Rewrite sub() to be more sensible, and easier to describe. Then a good description would be:

```
// returns true iff sequences A, B exist such that  
//   src = A : part : B  
// where ":" is sequence concatenation  
<T> boolean sub(List<T> src, List<T> part)
```

This is a decent specification

Mathematical flavor is not necessary, but can help avoid ambiguity

sneaky fringe benefit of specs #1

The discipline of writing specifications changes the incentive structure of coding

rewards code that is easy to describe and understand

punishes code that is hard to describe and understand (even if it is shorter or easier to write)

If you find yourself writing complicated specifications, it is an incentive to redesign

sub() code that does exactly the right thing may be slightly slower than hack that assumes no partial matches before true matches – but cost of forcing client to understand the details is too high

6.170 specifications

The “**precondition**”: constraints that hold before the method is called (if not, all bets are off)

requires – spells out any obligations on client

The “**postcondition**”: constraints that hold after the method is called (if the precondition held)

modifies – lists objects that may be affected by method; any object not listed is guaranteed to be untouched

throws – lists possible exceptions

effects – gives guarantees on the final state of modified objects

returns – describes return value

Example

public static int test(List lst, Object oldelt, Object newelt)

requires

modifies

effects

returns

```
public static int test(List lst, Object oldelt, Object
newelt) {
    for (int i = 0; ; i++) {
        if (lst.get(i) == oldelt) {
            lst.set(newelt, i);
            return i;
        }
    }
}
```

Example

public static int test(List lst, Object oldelt, Object newelt)

requires lst, oldelt and newelt are non null. oldelt occurs in lst

modifies

effects

returns

```
public static int test(List lst, Object oldelt, Object  
newelt) {
```

```
    for (int i = 0; ; i++) {
```

```
        if (lst.get(i) == oldelt) {
```

```
            lst.set(newelt, i);
```

```
            return i;
```

```
        }
```

```
    }
```

```
}
```

Example

`public static int test(List lst, Object oldelt, Object newelt)`

requires lst, oldelt and newelt are non null. oldelt occurs in lst

modifies lst

effects

returns

```
public static int test(List lst, Object oldelt, Object  
newelt) {
```

```
    for (int i = 0; ; i++) {
```

```
        if (lst.get(i) == oldelt) {
```

```
            lst.set(newelt, i);
```

```
            return i;
```

```
        }
```

```
    }
```

```
}
```

Example

public static int test(List lst, Object oldelt, Object newelt)

requires lst, oldelt and newelt are non null. oldelt occurs in lst

modifies lst

effects change the first occurrence of oldelt in lst to newelt
& makes no other changes to lst

returns

```
public static int test(List lst, Object oldelt, Object  
newelt) {
```

```
    for (int i = 0; ; i++) {
```

```
        if (lst.get(i) == oldelt) {
```

```
            lst.set(newelt, i);
```

```
            return i;
```

```
        }
```

```
    }
```

```
}
```

Example

public static int test(List lst, Object oldelt, Object newelt)

requires lst, oldelt and newelt are non null. oldelt occurs in lst

modifies lst

effects change the first occurrence of oldelt in lst to newelt
& makes no other changes to lst

returns an i such that ith element of lst was oldelt

```
public static int test(List lst, Object oldelt, Object  
newelt) {
```

```
    for (int i = 0; ; i++) {
```

```
        if (lst.get(i) == oldelt) {
```

```
            lst.set(newelt, i);
```

```
            return i;
```

```
        }
```

```
    }
```

```
}
```

Another Example

```
public static List<Integer> listAdd( List<Integer> lst1,  
                                     List<Integer> lst2)
```

requires ???

modifies ???

effects ???

returns ???

```
static List<Integer> listAdd( List<Integer> lst1,  
                              List<Integer> lst2) {  
  
    List<Integer> res = new ArrayList<Integer>();  
  
    for(int i = 0; i < lst1.size(); i++) {  
        res.add(lst1.get(i) + lst2.get(i));  
    }  
  
    return res;  
}
```

Another Example

```
public static List<Integer> listAdd( List<Integer> lst1,  
                                     List<Integer> lst2)
```

- requires** lst1 and lst2 are not null. lst1 and lst2 are the same size
 - modifies** none
 - effects** none
 - returns** a list of same size where the ith element is the sum of the ith elements of lst1 and lst2
-

```
static List<Integer> listAdd( List<Integer> lst1,  
                              List<Integer> lst2) {  
  
    List<Integer> res = new ArrayList<Integer>();  
  
    for(int i = 0; i < lst1.size(); i++) {  
        res.add(lst1.get(i) + lst2.get(i));  
    }  
  
    return res;  
}
```

Yet Another Example

```
static void listAdd2( List<Integer> lst1,  
                    List<Integer> lst2)
```

requires ???

modifies ???

effects ???

returns ???

```
static void listAdd2(List<Integer> lst1,  
                    List<Integer> lst2) {  
    for(int i = 0; i < lst1.size(); i++) {  
        lst1.set(i, lst1.get(i) + lst2.get(i));  
    }  
}
```

Yet Another Example

```
static void listAdd2( List<Integer> lst1,  
                    List<Integer> lst2)
```

requires lst1 and lst2 are not null. lst1 and lst2 are the same size
modifies lst1
effects ith element of lst2 is added to the ith element of lst1
returns none

```
static void listAdd2(List<Integer> lst1,  
                    List<Integer> lst2) {  
    for(int i = 0; i < lst1.size(); i++) {  
        lst1.set(i, lst1.get(i) + lst2.get(i));  
    }  
}
```

Yet Another (buggy) Example

public static void uniqueify (List<Integer> lst)

requires ???

modifies ???

effects ???

returns ???

```
public static void uniqueify(List<Integer> lst) {  
    for(int i=0; i < lst.size()-1; i++)  
        if(lst.get(i) == lst.get(i+1))  
            lst.remove(i);  
}
```

6.170 binarySearch

```
public static int binarySearch(int[] a, int key)
```

requires: a is sorted in ascending order

modifies: none

effects: none

returns:

some i such that $a[i] = \text{key}$ if such an i exists,
otherwise -1

Returning $\{ (\textit{insertion point}), -1 \}$ is very ugly, and an invitation to bugs and confusion; please read full specification and think about why the designers did this, and what the alternatives are. We'll return to the topic of special values in a later lecture.

should requires clause be checked?

If the client calls a method without meeting the precondition, the code is free to do anything, including pass corrupted data back

It is polite, nevertheless, to fail-fast: to provide an immediate error, rather than simply letting mysterious bad stuff happen

Preconditions used more in “helper” methods/classes rather than public libraries – friendlier to just deal with all possible input

Why does `binarySearch` impose a precondition rather than simply failing if list is not sorted?

Rule of Thumb: Check if cheap to do so

Ex: list has to be non-null → check

Ex: list has to be sorted → skip

comparing specifications

When is specification S1 weaker than S2?

When $\forall P, (P \text{ satisfies } S2) \Rightarrow (P \text{ satisfies } S1)$

Intuitively, we weaken a specification when we change it to give greater freedom to the implementor

We can weaken a specification by

Making requires harder to satisfy

Adding things to modifies clause

Making effects easier to satisfy

What is the strongest (most constraining) requires clause?

true

What is the strongest (most constraining) effects clause?

false

What is the strongest (most constraining) modifies clause?

modifies nothing



sneaky fringe benefit of specs #2

Specification means that client doesn't need to look at implementation

So code may not even exist yet!

Write specifications first, make sure system will fit together, and then assign separate implementors to different modules

Allows teamwork and parallel development (this is crucial, as you'll see towards the end of term)

Also helps with testing, as we'll see shortly